Aho – Corasick Algorithm in Pattern Matching

Elvan Owen and 13513082[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13513082@std.stei.itb.ac.id

***Abstract*— This paper is going to talk about Pattern Matching using Aho – Corasick algorithm which is quite useful in some cases where other algorithm like Knuth Morris Pratt or Boyer Moore or the other is just not fast enough.**

***Index Terms*— Automaton, Breadth – First Search, Suffix Tree, Pattern Matching.**

I. INTRODUCTION

What is Pattern Matching ?

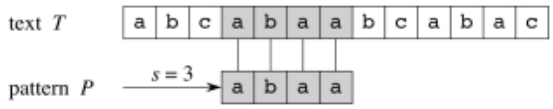
In computer science, pattern matching is the act of checking a given sequence of tokens (string) for the presence of the constituents of some pattern. Different from pattern recognition, the match commonly has to be exact. The patterns generally have the form of either sequences or tree structures. Uses of pattern matching include outputting the locations (if exist) of a pattern within a token sequence (string), to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (i.e., search and replace). Examples are given below.

Given :

T: text that is a (long) string with n characters

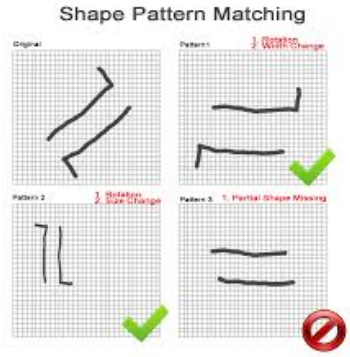
P: pattern that is a string with m characters that will be matched with the text

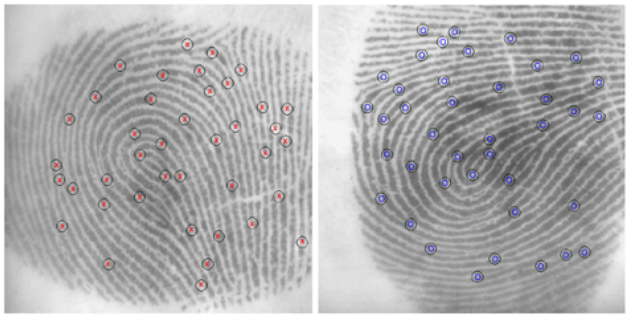
Find or Locate the first index in text which matched the pattern given.



Applications:

- Finding text in a text editor
- Web Search Engine
- Image Analysis
- BioInformatics





We have long used Pattern Matching Algorithm in our lives without us realizing it. For example, when we have read a book and then the next day when we refer back to the book, we forget which page we last read, but we know what we last read or in the case when we want to find a file in our directory, what is done behind the scenes is that the software or the Operating System has done the pattern matching for us transparently. In the end, there are a lists of pages or file names listed based on our search and we get or find what we intend to find.

II. BIOGRAPHY

Here I found a bit of biography about Professor Aho, but nothing about Corasick. Therefore, I just mention professor Aho.

Alfred V. Aho is the Lawrence Gussman Professor in the Department of Computer Science at Columbia University. He served as Chair of the department from 1995 to 1997, and again in the spring of 2003.

Professor Aho has a B.A.Sc in Engineering Physics from the University of Toronto and a Ph.D. in Electrical

Engineering/Computer Science from Princeton University.

Professor Aho won the Great Teacher Award for 2003 from the Society of Columbia Graduates. In 2014 he was again recognized for teaching excellence by winning the Distinguished Faculty Teaching Award from the Columbia Engineering Alumni Association.

Professor Aho has received the IEEE John von Neumann Medal and is a Member of the U.S. National Academy of Engineering and of the American Academy of Arts and Sciences. He is a Fellow of the Royal Society of Canada. He received honorary doctorates from the Universities of Helsinki and Waterloo, and is a Fellow of the American Association for the Advancement of Science, ACM, Bell Labs, and IEEE.

Professor Aho is well known for his many papers and books on algorithms and data structures, programming languages, compilers, and the foundations of computer science. His book coauthors include John Hopcroft, Brian Kernighan, Monica Lam, Ravi Sethi, Jeff Ullman, and Peter Weinberger.

Professor Aho is the "A" in AWK, a widely used pattern-matching language; "W" is Peter Weinberger and "K" is Brian Kernighan. (Think of AWK as the predecessor of perl.) He also wrote the initial versions of the string pattern-matching utilities egrep and fgrep that are a part of UNIX; fgrep was the first widely used implementation of what is now called the Aho-Corasick algorithm.

Professor Aho's current research interests include programming languages, compilers, algorithms, software engineering, and quantum computation.

Professor Aho has served as Chair of the Computer Science and Engineering Section of the National Academy of Engineering, as Chair of ACM's Special Interest Group on Algorithms and Computability Theory, and twice as Chair of the Advisory Committee for the National Science Foundation's Computer and Information Science and Engineering Directorate. He is currently the co-chair of the contributed and review articles sections of the Communications of the ACM.

Prior to his current position at Columbia, Professor Aho was Vice President of the Computing Sciences Research Center at Bell Labs, the lab that invented UNIX, C and C++. He was previously a member of technical staff, a department head, and the director of this center. Professor Aho also served as General Manager of the Information Sciences and Technologies Research Laboratory at Bellcore (now Telcordia).

Professor Aho plays bridge, golf, and the violin in a string quartet.

III. How it works

Aho – Corasick Algorithm idea is just as the other normal Pattern Matching algorithm like KMP which takes advantage of previous comparison to skip unnecessary comparison. This algorithm uses a trie data structure. Before getting into how the algorithm works. We'll talk about what trie is.

The term trie comes from the word "retrieval". Trie is a data structure where information retrieval is efficient. Using trie, search complexities can be brought to optimal limit (key length). If we use binary search tree to store keys, a well balanced BST will still need time proportional to (M * log N), where M is maximum string length and N is total keys in tree. Nonetheless, using trie, we can search the key in O(M) time. However, the penalty is on trie storage requirements which is quite a problem.

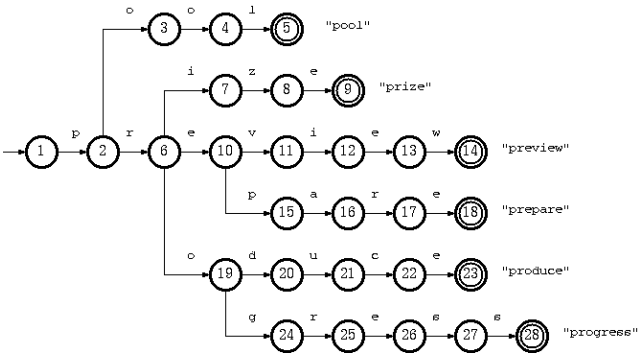
Each node in trie consists of multiple branches (all corresponding alphabets in the domain). We need to mark the last node of every key (by using a flag) to distinguish the node from the other node which is not the end of a key. A simple structure to represent nodes in English alphabet is as following,

```
struct trie_node
{
    int value; /* Used to mark leaf nodes */
    trie_node_t *children[ALPHABET_SIZE];
};
```

Inserting a new key into trie is easy. Each character of input key is inserted starting from the root as an individual trie node. Note that a node's children is an array of pointers to next level trie nodes. The key character acts as an index for the children's array. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark it as a leaf node. If the input key is a prefix of an already existing key, we simply mark the last node as a leaf node. The maximum key length determines the trie depth, which is quite a problem if we have a very long key.

Searching operation for a key is very similar to an insert operation, however we just compare each character in the key one by one and move down. The search can terminate due to end of string (marked as a leaf node) or lack of key in trie (reach the end of the trie).

Insert and search operation costs O(m) where m is the maximum length of a key, however the memory requirements of trie is O(ALPHABET_SIZE * m * n) where n is number of keys in trie. There are another efficient representation of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize the memory requirements of a trie.



Trie Data Structure

After knowing trie, the second thing to know before we dive into Aho – Corasick Algorithm is a suffix tree. What does suffix means ?

If Text=t1t2...ti...tn is a string, then Si=titi+1...tn is the suffix of Text that starts at position i and ends at position n.

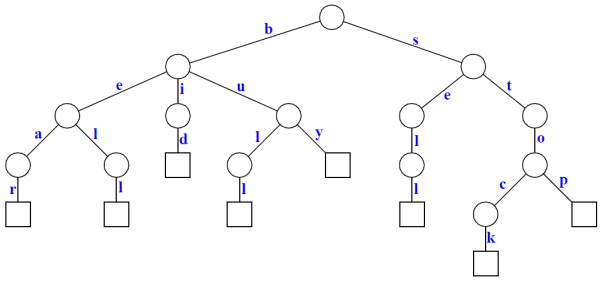
For example, Text = mississippi

S1 = mississippi
S2 = ississippi
S3 = ssissippi
S4 = sissippi
S5 = issippi
S6 = ssippi
S7 = sippi
S8 = ippi
S9 = ppi
S10 = pi
S11 = i
S12 = (empty)

A Suffix Tree for a given text is a compressed trie where the keys are all the suffixes of the given text. A suffix tree allows pretty fast implementations of many important string operations. Let's understand Compressed Trie with the following array of words.

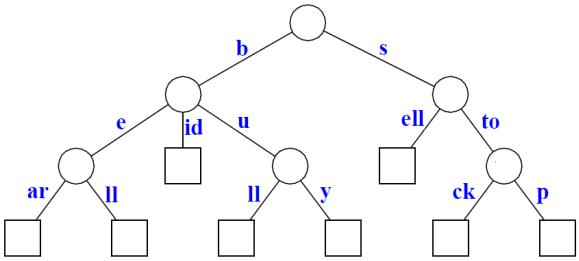
```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input.



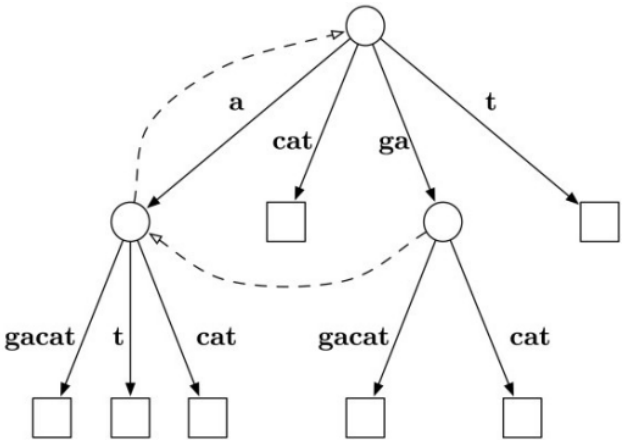
Standard Trie

Following is the compressed trie. A Compressed Trie is obtained from a standard trie by merging chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes in order to save spaces (memory).



Compressed Trie

One important feature of suffix trees are suffix links. Suffix links are links from a current node to another node which is a suffix of the current node. For example, if there is a node v in the tree with a label cå, where c is a character and å is a string (non-empty), then the suffix link of v points to a node with label å. If å is empty, then the suffix link of v is the root.



Suffix Tree with Suffix Links

Now it comes to the real part, the Aho – Corasick Algorithm. First of all, why would we want to use Aho – Corasick instead of another Pattern Matching Algorithm like KMP or Boyer Moore or Rabin Karp.

In the normal Pattern Matching algorithm, for any pattern from a set `P = {P1, . . . , Pk}`, to find the occurrence in a text `T[1 . . . m]`, it can be solved in time

```
O(|P1| + m + · · · + |Pk| + m) = O(n + km)
```

where `n` is the total length of all the pattern in `P`, by applying the algorithm for k times.

Aho-Corasick algorithm (AC) is a classic solution to this problem. It works in time `O(n + m + z)`, where `z` is number of pattern occurrences in `T`.

Steps in building Aho – Corasick Algorithm

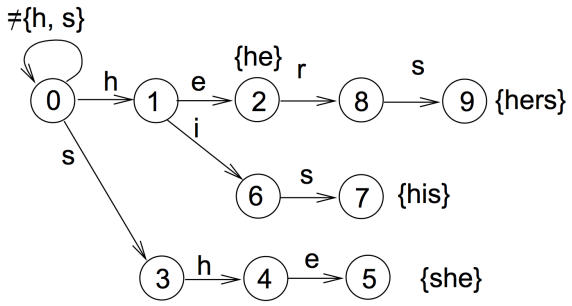
Building Aho Corasick Automaton

There are 2 phases :

Phase I:

1. Construct the keyword tree for *P*
   - for each $p \in P$ added to the tree, set *out(v)* = {*p*} for the node v labeled by *p*
2. complete the **goto function** for the root by setting
   $g(0, a) = 0$
   for each $a \in \Sigma$ that doesn't label an edge out of the root

If the alphabet Σ is fixed, Phase I takes time *O(n)*



Result of Phase I

Phase II:

After building the automaton, we can then build the longest-suffix-available-in-automaton links, which is the **failure function** *f*.

The algorithm is given as follows,

```
Q := emptyQueue();
for a ∈ Σ do
    if q(0, a) = q ≠ 0 then
        f(q) := 0; enqueue(q, Q);
while not isEmpty(Q) do
    r:= dequeue(Q);
    for a ∈ Σ do
        if g(r, a) = u ≠ ∅ then
            enqueue(u, Q); v := f(r);
            while g(v, a) = ∅ do v := f(v);
            f(u) := g(v, a);
            out(u) := out(u) ∪ out(f(u));
```

In this phase, what we have done is to do a *Breadth-First Search* starting from root 0, and in every level we create a link to the longest possible suffix existed in the graph if there is one, or else link it back to the root. While traversing the automaton, we append the values of the linked-node to the current-node so that when we end in the longer node for example *"she"* we could not only get *"she"* but also *"he"*.

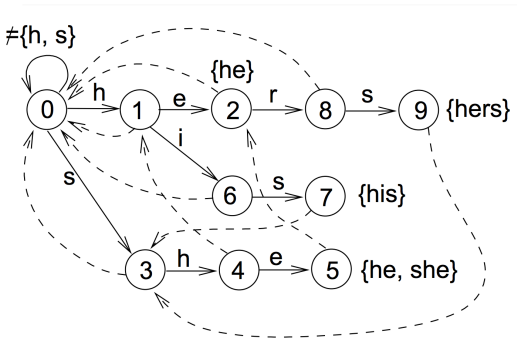
Using the Automaton to retrieve occurrence of Pattern in Text

States : nodes of the keyword tree
Initial state : 0 (the root)

Actions are determined by three functions:

1. **Goto function** *g(q, a)* gives the state entered from current state *q* by matching target char *a*
   - if edge *(q, v)* is labeled by *a*, then *g(q, a) = v*
   - *g(0, a)* = 0 for each *a* that does not label an edge out of the root the automaton stays at the initial state while scanning non-matching characters
   - Otherwise *g(q, a)* = ø
2. **Failure function** *f(q)* for *q* <> 0 gives the state entered at a mismatch
   - *f(q)* is the node labeled by the *longest proper suffix w* of *L(q)* such that *w* is a prefix of some pattern

   NB: *f(q)* is always defined, since *L(0)* = ε is a prefix of any pattern
3. **Output function** *out(q)* gives the set of patterns recognized when entering state *q*



Dashed arrows are fail transitions

Aho – Corasick Automaton

Below is the algorithm for Aho – Corasick,

```
q := 0; // initial state (root)
for i := 1 to m do
    while g(q, T[i]) = ∅ do
        q := f(q); // follow a fail
    q := g(q, T[i]); // follow a goto
    if out(q) ≠ ∅ then print i, out(q);
endfor;
```

Following steps by step from the algorithm :

1. We start from the root (*node 0*)
2. For each character in the Text, we move forward in the automaton, however if we are not able to move forward (due to unavailable character), then we have to use the **failure function** *f(q)* and go to the next node until we find a node in which it is possible to move forward with current character *T[i]*.
3. We move forward after finding a node through failure links.
4. Check if current node is a key node (node where it contains patterns from *P*)
5. Return to step 1 until no more characters left in *T*

Complexity of Aho – Corasick Algorithm

Theorem

Searching text *T[1 . . m]* with an Aho - Corasick automaton has *O(m + z) time complexity*, where z is the number of pattern occurrences .

Proof

For every character in text *T*, the automaton performs 0 or more **Fail function** *f* transitions, followed by a **Goto function** *g* .

Each *goto* either stays at the root, or increases the depth of *q* by at most 1 ⇒ the depth of *q* increased ≤ *m* times .

Each *fail* moves *q* closer to the root ⇒ the total number of *fail* transitions is ≤ *m*. So the maximum number of *fail* moves is equal to the maximum depth of *q* .

The *z* occurrences can be outputted in *z* × *O(1)* = *O(z) time complexity* .

So, the total Complexity ~ *O(m+z)* .

IV. Applications

In bioinformatics, where there are known sets of DNAs that need to be checked from a series of DNA.

Another one is that we can use this algorithm when we have *Wild Card* * character in our pattern that we want to search in the Text.

Let * be a *wild card* that matches any single character.

For example, `ab**c*` occurs at positions 2 and 7 of

```
12345678901 2
xabvccababca
```

If the number of wild cards is bounded by a constant, the patterns can be matched in linear time.

Let $P = \{P_1, \ldots, P_k\}$ be the set of patterns which is substrings separated by wild-cards, and let $l_1, \ldots, l_k$ be their end positions (index) in *T*

Preprocess:

1. Build automaton for all the patterns in P
2. Zero all occurrence counts :

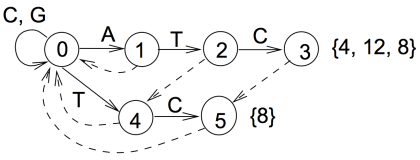
```
for i = 1 to |T| do C[i] = 0
```

Search:

1. Search text *T* using the automaton
2. When pattern P_j is found to end at index *i* of *T*, increment $C[i - l_j + 1]$ by one
3. Any index *i* with $C[i] = k$ is a start position of an occurrence

Example :

Let $P = \phi ATC\phi\phi TC\phi ATC$
Then $\mathcal{P} = \{ATC, TC, ATC\}$ with $l_1 = 4$, $l_2 = 8$ and $l_3 = 12$



Search on

```
i: 12345678901234...
T: ACGATCTCTCGATC...
```

⇝ $C[1] = C[7] = C[11] = 1$ and $C[3] = 3$ (~ occurrence)

V. Conclusion

By using Aho – Corasick Algorithm, we can have a lot of improvement in performance especially in *time complexity* when we have a set of patterns *P* we need to match to a text *T*.

VI. Acknowledgment

Terima kasih kepada Bu Ulfa dan Bapak Rinaldi yang telah membimbing saya selama satu semester ini. Tanpa mereka, saya tidak akan dapat menulis makalah ini.

REFERENCES

[1] http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf.
[2] http://www.geeksforgeeks.org/trie-insert-and-search/.
[3] http://linux.thai.net/~thep/datrie/trie1.gif.
[4] http://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/.
[5] http://www.cbcb.umd.edu/confcour/CMSC858W-materials/lecture5.pdf.
[6] http://www1.cs.columbia.edu/~aho/bio.html
[7] http://open-your-innovation.com/2010/12/21/matching-algorithm-and-process-fail-to-engage-solvers-into-problem-solving/
[8] http://i.stack.imgur.com/uh9O1.png

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 May 2015

Elvan Owen
13513082